Diagramming Program Values by Spatial Refinement

SIDDHARTHA PRASAD, Brown University, USA
MICHAEL TU, Brown University, USA
KARAN KASHYAP, Brown University, USA
TIM NELSON, Brown University, USA
SHRIRAM KRISHNAMURTHI, Brown University, USA

Diagrams enable programmers to reason, debug, and communicate. However, constructing diagrams for programming language data is unnecessarily hard. We present a declarative DSL, Spytial, that captures the essential *spatial* features of data. We endow Spytial with a spatial semantics, mapping values to the 2D plane, and prove key properties. Spytial uses constraint-solving to make interactive renderings. We show how Spytial can be embedded in three very different languages: Python, Rust, and Pyret. We present a novel counterfactual debugging aid for diagramming errors, combining textual and visual output. We evaluate the language and system for expressiveness, performance, and diagnostic quality. Finally, we also show how Spytial can be used to *construct* values interactively and visually while preserving spatial constraints.

CCS Concepts: • **Software and its engineering** → **Specification languages**; *Constraint and logic languages*; Semantics; • **Human-centered computing** → *Visualization toolkits.*

Additional Key Words and Phrases: diagramming, visualization, constraint solving, counterfactuals, DSL

ACM Reference Format:
Siddhartha Prasad, Michael Tu, Karan Kashyap, Tim Nelson, and Shriram Krishnamurthi. 2026. Diagramming Program Values by Spatial Refinement. *Proc. ACM Program. Lang.* 10, PLDI, Article 197 (June 2026), 25 pages. https://doi.org/10.1145/3808275

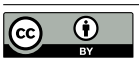
1 Introduction

Human reasoning is profoundly spatial. Cognitive science shows that people use space to structure thought about time, causality, and abstract relations [47]. Diagrams exploit this fact: placing nodes above, beside, or within one another reveals structure and reduces cognitive effort [20].

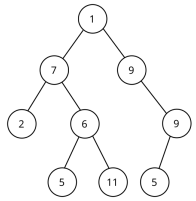
Space is all over our programs. Sometimes it is *inherent* to the domain, such as a chess board. Sometimes it is imposed by *convention*: trees grow downward and BDDs are layered structures (see figs. 1 and 3a). These shared spatial models are not merely decorative. Diagrams that follow these conventions enhance reasoning and debugging; ones that don't can be confusing, even misleading the reader if they create the kind of incongruence captured by the Stroop Effect [42, 50].

Despite their utility, diagramming data is often an afterthought in programming. Generating diagrams can require significant effort: distracting from the primary goal, demanding knowledge of drawing libraries, requiring knowledge of drawing intricacies (such as rules on how to render visual objects (§9)), and more. Thus, while diagrams abound in books and tutorials, implementations do not enjoy the same support. We state with great precision what a tree, queue, buffer, or BDD *is*, but give no hint as to how it should *look*.

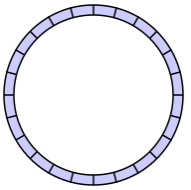
Authors' Contact Information: Siddhartha Prasad, Brown University, USA; Michael Tu, Brown University, USA; Karan Kashyap, Brown University, USA; Tim Nelson, Brown University, USA; Shriram Krishnamurthi, Brown University, USA.



ACM 2475-1421/2026/6-ART197
https://doi.org/10.1145/3808275



(a) Binary tree as a vertical hierarchy [36].



(b) Circular buffers are conceived as rings [10].

Fig. 1. Diagrams of common data structures from Wikipedia.

This paper presents a programming language, Spytial, which has a *spatial* semantics: programs denote how data are rendered as graphs in (2D) space. We observe that every language already provides a default representation of values through inspection, printing, or serialization, which expose a value's structure as a graph of nodes and edges. Structured values can, therefore, be given denotations as sets of possible layouts in two dimensions. Rules in Spytial declaratively refine this set of renderings. *Constraints* restrict the set of renderings (forcing nodes to be above, below, etc. others); an empty set of possible renderings indicates a *spatial* error. *Directives* alter presentation (color, iconography, etc.). This semantics enables the incremental creation of Spytial programs, and makes it easy to selectively turn rules on and off. Furthermore, Spytial lets users interactively readjust the rendering, and preserves constraints and directives as they do.

Spytial is likely to have two kinds of users: authors of libraries and their consumers. Typically, we expect library authors to provide visualizations that render their data in some useful, perhaps canonical, form. Consumers would primarily use these visualizations to understand the data structures and to identify when they have created subtly invalid instances.

However, the roles are not mutually exclusive. We expect that Spytial is especially useful during software development (and accompanying debugging). During that time, the user frequently switches between these two roles. Similarly, library data structures are often used in domain-specific ways: either enriching the structure (e.g., augmenting a binary search tree with height information to get an AVL tree) or instantiating it in a specific context (e.g., treating a generic queue as a collection of tasks in a job scheduler). The consumer may then want to enrich the visualization (e.g., align nodes by height or distinguish tasks by color), thereby becoming an author.

Another useful way to think about Spytial is as enhancing read-eval-print-loop (REPL) output. REPLs are invaluable while developing and debugging programs, but their output remains fundamentally linear and textual, making even modestly-sized values difficult to interpret. Pretty-printers use indentation to convey *very* limited spatial structure. We view Spytial as a significant step forward: just as languages have a "`toString`" operation, Spytial conceptually provides a "`toDiagram`": a richer, interactive visual presentation of data that enriches the REPL.

We introduce Spytial in §s 2 and 3 and then make the following contributions:

(1) *Spatial semantics* (§4): We formalize how program values can be interpreted as sets of possible layouts, with constraints and directives refining them.
(2) *Managing unsatisfiable constraints* (§5): What happens when constraints conflict? Spytial could just halt with an error. Instead, it constructs a counterfactual layout that explains the constraint conflicts. This contribution is independent of Spytial and could be applied to other solver-based tools that show instances visually (e.g., Alloy [18]).
(3) *Integration into languages* (§6): We implement Spytial for three very different languages (Python, Rust, Pyret) to analyze commonalities and differences and inform adoption.

(4) *Evaluation* (§7): We assess Spytial's expressiveness, interaction speed, and visualization of unsatisfiability. We also report on preliminary use.
(5) *Constraint-driven input* (§8): We also show that, because declarative rules can be "run backward", Spytial programs can drive a prototype structured editor for *constructing* terms in a way that preserves the spatial semantics.

Supplementary Materials. The supplement includes high-resolution and interactive versions of all figures. The mark "◇ supplement" indicates other supplementary content including the Lean formalization, implementations, Jupyter notebooks, and user study details; see https://zenodo.org/records/19520876.

2 Running Example: Binary Decision Diagrams

To demonstrate Spytial, we incrementally work through the development of binary decision diagrams (BDDs) as a running example. Imagine you are building a BDD library and want to debug or visualize the structures it produces.

Suppose we create the BDD for the formula $(\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2) \vee (x_2 \wedge x_3)$ (as shown in fig. 3a). Most programmers' inspection practices never get much beyond "printf-style" debugging [6]. For a structure like a BDD, that means seeing a constructor-level representation in which each node is shown only through its fields and references to children:

```
Node(15, 'x1', Node(14, 'x2', Node(8, 'x3', TRUE, FALSE), Node(4, 'x3',
    ↪ FALSE, TRUE)), Node(3, 'x2', FALSE, TRUE))
```

This output is typical of default inspection and serialization: it exposes recursive structure as nested text, but leaves the programmer to mentally reconstruct the BDD's branching and organization.

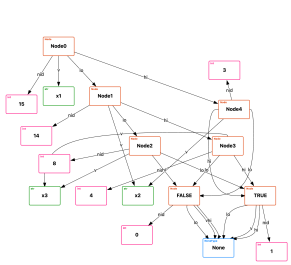
A library author may go further and provide both themselves and their users with a structural view where the branching structure is directly visible (perhaps resembling fig. 3a). The Python package dd for BDDs (https://github.com/tulip-control/dd) illustrates a common pattern: export the structure to DOT [14] and render it as a diagram. We compare against it below.

Spytial pursues the same end, but without asking users (or library authors) to build a separate export pipeline. Instead, it derives a lightweight diagram directly from ordinary structured values: data values become graph nodes (with different colors for each type) and fields become edges between them. This is shown in fig. 2a. Nodes representing primitive values (e.g., string `"x1"`) are labeled by their values, while those representing constructs with previously used variable names (e.g., Nodes TRUE and FALSE) are labeled by those names. Other objects (e.g., Node0, Node1) are labeled by a deterministic type-based fallback. While this default diagram is quite far removed from the visualization in fig. 3a, it gives the user a starting point from which to determine what to improve. (It is worth noting that the figure already shows the sharing of the nodes representing truth and falsity, a detail lost in Python's textual output.) The Spytial philosophy is to then improve it incrementally:

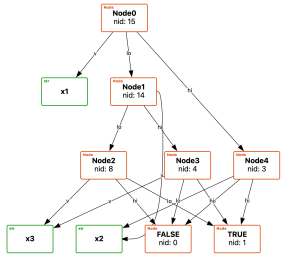
(1) Move the node identifier (`nid`) from an edge label to a variable attribute, and hide non-informative atom types. This is shown in fig. 2b, which is produced by this Spytial code:

```
attribute(field="nid")
hideAtom(selector="NoneType+int")
```

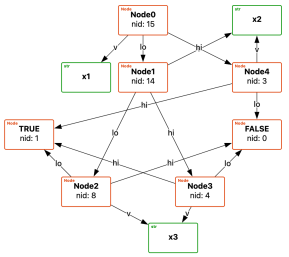
In dd, achieving this kind of customization would typically require modifying the DOT export itself or writing post-processing code over the emitted graph. In practice, that makes the customization the responsibility of the library author (or of a sufficiently motivated consumer). In Spytial, the same change is a small declarative rule available to either author or consumer.



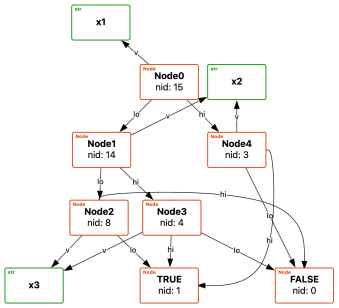
(a) Default Spytial output.



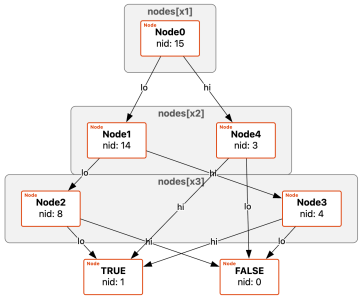
(b) Hide/collapse impl details.



(c) Align by tested variable.



(d) Order layers by variable order.



(e) Group nodes by variable.

Fig. 2. Progressive refinement of the Spytial BDD diagram. Large versions are in the ◇ supplement.

(2) Layer by tested variable. Nodes testing the same variable occupy one horizontal layer. The further-refined figure is shown in fig. 2c, which is produced by adding

```
align(selector="{x,y : Node | (x != y) and (x.v) = (y.v)}", direction="horizontal")
```

With dd's DOT export, this layout logic would typically be embedded in the code that generates the graph, again placing it largely in the hands of the export author. Here it appears instead as a separate declarative constraint that either an author or a consumer can add or remove.

(3) Order layers top-to-bottom by variable order. The further-refined figure is shown in fig. 2d, and is produced by

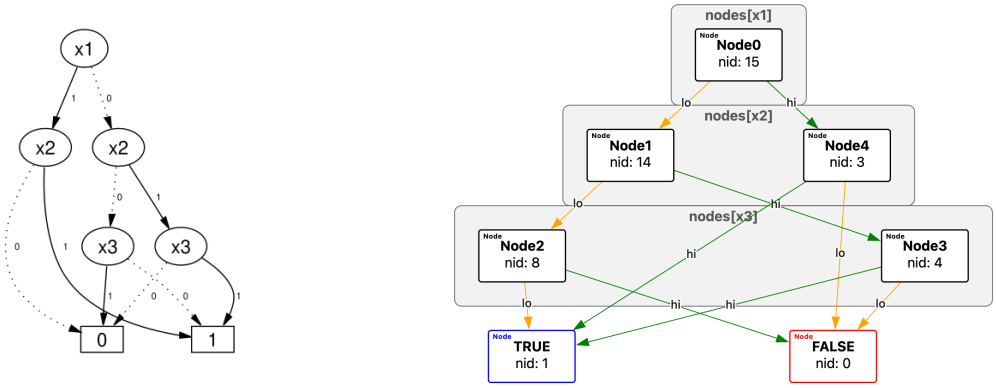
```
orientation(selector="{x, y : Node | x->y in (lo + hi)}", directions=["below"])
```

Here these layout conventions are explicit semantic constraints rather than incidental properties of a particular dd export routine. They are specified independently of the code that constructs the diagram, and are therefore open to both library authors and users.

(4) Group by variables. Represent variables as groups of nodes rather than as isolated boxes. The resulting figure is shown in fig. 2e and is produced by

```
group(selector="{vr : str, y : Node | @:(vr) = @:(y.v)}", name="nodes")
hideAtom(selector= "str")
```

Expressing such grouping in dd would require changing the exported DOT code, e.g., by adding nodes or cluster annotations. In Spytial, authors and consumers can instead express grouping as a separate rule over the underlying data, refining views incrementally without modifying the DOT export or maintaining a separate post-processing pipeline. Spytial thus shifts control over presentation from the library alone to downstream users as well.



(a) BDD diagram from Wikipedia [7].

(b) Final Spytial BDD diagram.

Fig. 3. BDD diagrams from Wikipedia and Spytial. Both depict BDDs as layered structures: variables ordered top-to-bottom, nodes testing the same variable aligned horizontally, and equivalent nodes shared.

A handful of Spytial constraints thus suffice to reflect the essence of a BDD's diagram. Directives provide polish and encode conventions:

(1) Color nodes to distinguish non-terminals from terminals, and truth from falsity:

```
atomColor(value='red',selector="{x: Node | @num:(x.id) = 0}")
atomColor(value='blue',selector="{x: Node | @num:(x.id) = 1}")
atomColor(value='black',selector="{x: Node | (@num:(x.id) != 0) and (@num:(x.id) != 1)}")
```

(2) Style low and high branches to make them stand out better:

```
edgeColor(field="hi", value='green')
edgeColor(field="lo", value='orange')
```

(3) Unlike the Wikipedia figure, simplify traversal by assigning predictable directions to branches: low edges point left, high edges point right (if not pointing to terminals).

```
orientation(directions=["left"],
   selector="{x,y: Node | x->y in lo and (@num:(y.id) != 0) and (@num:(y.id) != 1)}")
orientation(directions=["right"],
   selector="{x,y : Node | x->y in hi and (@num:(y.id) != 0) and (@num:(y.id) != 1)}")
```

Spytial thus moves from raw structure to task-specific presentation. Figure 3b shows the final result, which (unlike the Wikipedia version) is an actual datum, with information such as node IDs that remains useful to a developer. It is important to note:

- The Spytial rules apply to *any* BDD. This makes the Spytial program a useful accompaniment to the BDD library for both developers and users.
- The rules are declarative, with no scope or ordering inter-dependencies. A user can thus easily add, remove, enable, and disable rules to explore the diagramming space. In contrast, comparable refinements in dd are often most naturally expressed by changing the export itself, which places more control with the library author than with the consumer.
- Spytial diagrams are *interactive*. Unlike diagrams rendered from dd's DOT output, whose layouts are static, Spytial diagrams let users move nodes while preserving the stated constraints. Readers can try this in the ◇ supplement.

$$
\begin{array}{rl}
\textit{Constraint} ::= & \text{orientation} \mid \text{cyclic} \mid \text{align} \\
& \mid \text{group} \mid \text{hideatom} \mid \text{size} \\
\textit{orientation} ::= & \text{Sel}_2 \times [\textit{direction}] \\
\textit{direction} ::= & \text{above} \mid \text{below} \mid \text{left} \mid \text{right} \\
& \text{directlyAbove} \mid \text{directlyBelow} \\
& \mid \text{directlyLeft} \mid \text{directlyRight} \\
\textit{cyclic} ::= & \text{Sel}_2 \times \textit{rotation} \\
\textit{rotation} ::= & \text{clockwise} \mid \text{counterclockwise} \\
\textit{align} ::= & \text{Sel}_2 \times \textit{aligndir} \\
\textit{aligndir} ::= & \text{horizontal} \mid \text{vertical} \\
\textit{group} ::= & \text{Sel}_2 \times \textit{name} \times \textit{showedge} \\
& \mid \text{Sel}_1 \times \textit{name}
\end{array}
$$

$$
\begin{array}{rl}
\textit{size} ::= & \text{Sel}_1 \times \textit{width} \times \textit{height} \\
\textit{hideatom} ::= & \text{Sel}_1 \\
\textit{Directive} ::= & \text{icon} \mid \text{atomcolor} \mid \text{edgecolor} \\
& \mid \text{attribute} \mid \text{hidefield} \\
& \mid \text{inferrededge} \\
\textit{icon} ::= & \text{Sel}_1 \times \textit{path} \times \textit{showLabels} \\
\textit{atomcolor} ::= & \text{Sel}_1 \times \textit{value} \\
\textit{edgecolor} ::= & \text{Sel}_1 \times \textit{field} \times \textit{value} \\
\textit{attribute} ::= & \text{Sel}_1 \times \textit{field} \\
\textit{hidefield} ::= & \text{Sel}_1 \times \textit{field} \\
\textit{inferrededge} ::= & \text{Sel}_2 \times \textit{name} \times \textit{color}
\end{array}
$$

Fig. 4. Abstract syntax of Spytial programs.

- Sometimes the constraints may not be realizable. We address this in §5.1. Unlike a DOT-based rendering pipeline, which may still produce a plausible-looking diagram for a malformed or inconsistent structure, Spytial treats such cases as unsatisfiable specifications and surfaces the inconsistency directly.
- The same rules can be used to *construct* BDDs! See §8. By contrast, dd exports are one-way pipelines that only produce renderings of an already-built BDD.

3 The Spytial Language

Figure 4 presents the abstract syntax of Spytial programs, divided into **constraints**, which specify placement, and **directives**, which control appearance. Both operate not on program values directly, but on the elements identified by **selectors**.

Spytial is inspired by, and draws ideas from, recent diagramming systems: Penrose [51], Bluefish [31], and Cope-and-Drag [32]. From Penrose and Bluefish, it borrows the idea of a relational substrate and relational mechanisms to identify which elements a visual rule should affect. From Cope-and-Drag it borrows constraints and directives, which it extends significantly. In particular, Cope-and-Drag is based on both significant cognitive science principles and corpus studies; these justify our adoption of its ideas. However, Spytial also represents significant improvements over these systems. We discuss the relationships in detail in §9.

Selectors operate over a relational model of program structure. Rather than introducing a new ad hoc relational language, Spytial implements the language of Alloy [18], which (a) has a clean relational semantics, (b) has a syntax that tries to mirror standard mathematical notation, and (c) is already familiar to many. This specific choice is, however, not essential to the design of Spytial; any relational language could work, and we have a prototype using SQL.

A key property of Spytial, implicit in §2 and explicit in the abstract syntax, is that it does not have variable bindings. This design decision, combined with its semantics-by-conjunction (§4.4), makes each rule truly independent of the others. This makes it easy for a user to turn each individual rule on and off without causing cascading errors.

4 Formalizing Spytial

Spytial maps host language values to diagrams realized on the 2D plane. Each value in the host language corresponds to an *atom*, and structural links (e.g., fields, arguments, pointers) become *relations* between them. In a diagram, atoms become *boxes* in the plane, and relations *arrows*

$$\text{For } \mathrm{s} \in \mathrm{Sel}_2 \qquad \frac{\forall (a \times b) \in [\![\mathrm{s}]\!].\ R(a).x_{tl} + R(a).width < R(b).x_{tl}}{R \models \mathrm{orientation}(\mathrm{s}, \mathrm{left})}\ (\mathrm{OL})$$

$$\frac{\forall (a \times b) \in [\![\mathrm{s}]\!].\ \mathrm{aligned}_{\mathrm{v}}(R(a),\ R(b))}{R \models \mathrm{align}(\mathrm{s}, \mathrm{vertical})}\ (\mathrm{OVA})$$

Fig. 5. Some judgement rules for the orientation and align constraints.

between those boxes. A *realization* is a partial function assigning atoms to boxes:

$$\mathrm{Box} = \{\, x_{tl}, y_{tl}, width, height : \mathbb{Q} \,\} \qquad R : \mathrm{Atom} \rightharpoonup \mathrm{Box}.$$

($\mathbb{Q}$ instead of $\mathbb{Z}$ enables better performance.) If $R(a)$ is defined, then a is visualized by the box $R(a)$. We write $\mathcal{R}$ for the set of all realizations over a fixed set of atoms, and $\mathcal{R}_{\mathrm{wf}} \subseteq \mathcal{R}$ for the set of *well-formed realizations*, namely those R such that every box has a positive width and height, no two boxes overlap, and distinct atoms map to distinct boxes (or are hidden).

Spytial specifications are written over the host program's types. Selectors (§4.1) map these static specifications to dynamic data by indicating what values or parts of a value a rule applies to. Constraints (§4.2) refine where those entities may appear by narrowing the space of admissible layouts. Directives (§4.3) refine how those parts are presented, controlling visual features such as color, iconography, and labels without affecting spatial arrangement.

4.1 Selectors

Spytial programmers describe rules that apply to whole classes of values. Selectors identify which specific runtime values—atoms or pairs of atoms—a rule should apply to. Formally, we distinguish between *Unary* selectors (Sel_1), which denote sets of individual atoms, and *binary* selectors (Sel_2), which denote sets of ordered pairs of atoms.

$$[\![\mathrm{Sel}_1]\!] := \{\, Atom \,\}_{\mathrm{fin}} \qquad [\![\mathrm{Sel}_2]\!] := \{\, \mathrm{Atom} \times \mathrm{Atom} \,\}_{\mathrm{fin}}$$

Selectors serve as the parameters to both directives and constraints (§s 4.2 and 4.3).

4.2 Constraints

Each Spytial constraint refines the meaning of a program by restricting the set of realizations that count as valid. Formally, these relationships are expressed as satisfaction judgements of the form $R \models C$, stating that the realization R satisfies the constraint C. To save space, only some representative rules are shown in fig. 5; the remainder, covering all constraints, are available in the Lean formalization given in the ◇ supplement.

As these rules suggest, every rule ultimately reduces to relationships among the coordinates and footprints of boxes. The essential work of a constraint is to compare, align, or bound these numeric quantities. For this reason, each rule can be compiled into a system of linear (in)equalities over the elements of Box, capturing the same semantic content in a solver-executable form. Spytial also seeds the solver with constraints to ensure that layouts are well-formed (e.g., boxes do not overlap, have positive width and height, etc.). Table 1 summarizes the translation from abstract syntax to inequality systems. To save space, a handful of similar rules have been omitted. The full ruleset is in the ◇ supplement.

4.2.1 Alignment Constraints. Alignment constraints require that selected boxes align along a common axis. For instance, a vertical alignment requires that all selected boxes share a common x coordinate at their top left corner.

Table 1. Translation of core Spytial constraints into systems of linear inequalities.

Abstract syntax	Emitted inequalities
orientation(s_2, left)	$\forall (a,b) \in R([\![s_2]\!]).\ a.x_{tl} + a.width < b.x_{tl}$
orientation(s_2, directlyLeft)	$\forall (a,b) \in R([\![s_2]\!]).\ a.x_{tl} + a.width < b.x_{tl} \land a.y_{tl} = b.y_{tl}$
orientation(s_2, above)	$\forall (a,b) \in R([\![s_2]\!]).\ a.y_{tl} + a.height < b.y_{tl}$
orientation(s_2, directlyAbove)	$\forall (a,b) \in R([\![s_2]\!]).\ a.y_{tl} + a.height < b.y_{tl} \land a.x_{tl} = b.x_{tl}$
cyclic(s_2, clockwise)	$\mathcal{P}(s_2) := \{\langle v_0,\ldots,v_k\rangle \mid k \geq 0,\ (v_i, v_{i+1}) \in [\![s_2]\!],\ v_0,\ldots,v_k \text{ distinct}\}.$
	$X \sqsubseteq Y := \exists j.\ 0 \leq j \leq \lvert Y\rvert - \lvert X\rvert \ \land\ \forall i < \lvert X\rvert.\ X[i] = Y[i+j],$
	$X \sqsubset Y := X \sqsubseteq Y \ \land\ \lvert X\rvert < \lvert Y\rvert.$
	$\forall [a_0,\ldots,a_{n-1}] \in \{ M \in \mathcal{P}(s_2) \mid \nexists N \in \mathcal{P}(s_2).\ M \sqsubset N \}.$
	Let $\theta_m^k \equiv \frac{2\pi}{n}(m+k);$
	$\mathsf{hrel}(i,j,k) \equiv \begin{cases} a_i.x_{tl} = a_j.x_{tl} & \text{if } \cos\theta_i^k = \cos\theta_j^k \\ a_i.x_{tl} + a_i.width < a_j.x_{tl} & \text{if } \cos\theta_i^k < \cos\theta_j^k \\ a_j.x_{tl} + a_j.width < a_i.x_{tl} & \text{otherwise.} \end{cases}$
	$\mathsf{vrel}(i,j,k) \equiv \begin{cases} a_i.y_{tl} = a_j.y_{tl} & \text{if } \sin\theta_i^k = \sin\theta_j^k \\ a_i.y_{tl} + a_i.height < a_j.y_{tl} & \text{if } \sin\theta_i^k > \sin\theta_j^k \\ a_j.y_{tl} + a_j.height < a_i.y_{tl} & \text{otherwise.} \end{cases}$
	$\bigvee_{k=0}^{n-1} \Big(\forall\, 0 \leq i < j < n.\ \mathsf{hrel}(i,j,k) \ \land\ \mathsf{vrel}(i,j,k) \Big)$
group(s_1, n)	$g_{\mathsf{n}} \in \{ x_{tl}, y_{tl}, x_{br}, y_{br} : \mathbb{Q} \}.$
	$\forall a \in R([\![s_1]\!]).\ (g_{\mathsf{n}}.x_{tl} \leq a.x_{tl}) \ \land\ (g_{\mathsf{n}}.y_{tl} \leq a.y_{tl}) \ \land$
	$(a.x_{tl} + a.width \leq g_{\mathsf{n}}.x_{br}) \ \land\ (a.y_{tl} + a.height \leq g_{\mathsf{n}}.y_{br})$
	$\forall b \in \mathsf{Box} \setminus R([\![s_1]\!]).\ (b.x_{tl} < g_{\mathsf{n}}.x_{tl}) \ \lor\ (b.y_{tl} < g_{\mathsf{n}}.y_{tl}) \ \lor$
	$(b.x_{tl} + b.width > g_{\mathsf{n}}.x_{br}) \ \lor\ (b.y_{tl} + b.height > g_{\mathsf{n}}.y_{br})$
group(s_2, n, addedge)	$\forall (a \times b) \in R([\![s_2]\!]).\quad g_{\mathsf{n}}^a \in \{ x_{tl}, y_{tl}, x_{br}, y_{br} : \mathbb{Q} \}$
	$(g_{\mathsf{n}}^a.x_{tl} \leq b.x_{tl}) \ \land\ (g_{\mathsf{n}}^a.y_{tl} \leq b.y_{tl}) \ \land\ (b.x_{tl} + b.width \leq g_{\mathsf{n}}^a.x_{br})$
	$\land\ (b.y_{tl} + b.height \leq g_{\mathsf{n}}^a.y_{br})$
	$\forall c \in \mathsf{Box} \setminus \{ b \mid (a \times b) \in R([\![s_2]\!]) \}.\quad g_{\mathsf{n}}^a \in \{ x_{tl}, y_{tl}, x_{br}, y_{br} : \mathbb{Q} \}$
	$(c.x_{tl} < g_{\mathsf{n}}^a.x_{tl}) \ \lor\ (c.y_{tl} < g_{\mathsf{n}}^a.y_{tl}) \ \lor$
	$(c.x_{tl} + c.width > g_{\mathsf{n}}^a.x_{br}) \ \lor\ (c.y_{tl} + c.height > g_{\mathsf{n}}^a.y_{br})$
align(s_2, horizontal)	$\forall (a,b) \in R([\![s_2]\!]).\ a.y_{tl} = b.y_{tl}$
align(s_2, vertical)	$\forall (a,b) \in R([\![s_2]\!]).\ a.x_{tl} = b.x_{tl}$
size(s_1, w, h)	$\forall a \in [\![s]\!].\ a.\mathsf{width} = w \ \land\ a.\mathsf{height} = h$

4.2.2 Orientation Constraints. Orientation constraints specify directional relationships between selected boxes. They come in looser—e.g., left, above—and stricter—e.g., directlyLeft, directlyAbove—forms. The stricter forms require alignment along the dimension not named in the constraint.

4.2.3 Grouping Constraints. Grouping constraints define a rectangular boundary such that all boxes in the group lie within the rectangle and all those not in the group lie outside it. There are two kinds of grouping constraints. Unary groupings, using Sel_1, introduce a single boundary

encompassing the selected boxes. Binary groupings, using Sel_2, interpret the first element of each pair as a *key* and the second as a *member*. Each distinct key introduces its own group boundary, which must contain exactly the boxes corresponding to its members. Binary groupings also support an addedge flag, which dictates whether to draw an edge from each key atom to its corresponding group boundary. To maintain coherence among groups, Spytial also imposes a subsumption discipline: if two groups overlap, then one must fully contain the other.

In the semantics, these group boundaries are existentially quantified. In the implementation, they are found by the solver. Thus, group constraints introduce new non-program-value geometric objects into the layout.

4.2.4 Cyclic Constraints. Cyclic constraints arrange a sequence of boxes in a polygonal pattern. Whereas orientation and alignment constraints compare pairs of boxes directly, cyclic constraints describe entire ordered sequences of boxes derived from a Sel_2, interpreted as an adjacency relation. From this relation, the semantics extracts simple directed paths and retains only those that are maximal under contiguous-subpath subsumption (intuitively, cycles or chains that cannot be extended further).

Each path defines a notional regular n-gon that captures the expected cyclic order of boxes. From this imagined arrangement, we derive relative position and alignment relations, expressed as inequalities in Box. Since only relative position matters, the constraint is satisfied if there *exists* an assignment of boxes to the vertices of this n-gon in the specified direction (clockwise or counterclockwise) that also respects the order of the directed path, for which all corresponding relative position and alignment relations hold.

In practice, this is realized by generating multiple families of inequalities: one for each of the n path-order preserving assignments of boxes to n-gon vertices. The cyclic constraint is satisfied if any one of these families is satisfied, introducing disjunctions in the generated system of inequalities.

4.2.5 Hiding and Size. A hideatom constraint removes selected atoms from the realization, making R a *partial* function from atoms to boxes. By changing which boxes appear, hiding alters the domain over which all other constraints are evaluated. Because hidden atoms are removed before constraint emission, hideatom rules do not produce inequalities themselves; they prune the relationalization prior to solving.

$$\frac{\mathrm{s} \in \mathrm{Sel}_1 \qquad \forall a \in [\![\mathrm{s}]\!].\ R(a) \text{ is undefined}}{R \models \mathrm{hideatom}(\mathrm{s})} \ (\textsc{HideAtom})$$

A size *sets* the width and height of selected boxes. Together, hiding and size determine *which* boxes are realized and *how large* they are, thereby influencing the domain and geometry of valid layouts.

4.3 Directives

Directives control how nodes and edges are rendered. They allow diagrammers to highlight important aspects of a diagram and introduce interpretive scaffolding. These directives alter the visual encoding of atoms and edges without changing the underlying structure. The abstract syntax of directives is shown in fig. 4. We summarize their meaning informally here:

- atomcolor — assign a color to the boxes representing the selected atoms.
- edgecolor — color edges originating from selected atoms for a specified field/name.
- attribute — replace outgoing edges (fields) from selected atoms with text labels.
- inferrededge — draw additional edges to expose derived or implicit relationships.
- icon — attach a graphical icon to selected atoms. By default, the icon replaces the box text; an optional flag can show both icon and label.
- hidefield — suppress drawing of edges for a given field name on selected atoms.

All directives are presentational: they change rendering only and do not affect the set of satisfying realizations. However, while they do not impact the spatial semantics, they are crucial to achieve some of the cognitive benefits of diagrams (§9).

4.4 Program Semantics

A Spytial program is a finite set of constraints and directives, which are composed using set union:

$$\text{Program} \;:=\; \{\,\text{Constraint}\,\}_{\text{fin}} \cup \{\,\text{Directive}\,\}_{\text{fin}}, \qquad \forall P, Q \in \text{Program},\;\; P \circ Q \;:=\; P \cup Q$$

A realization satisfies a program if it satisfies every constraint and directive in it. The denotation of a program is the set of well-formed realizations consistent with all of its constraints and directives:

$$[\![P]\!] := \{\, R \in \mathsf{R}_{\text{wf}} \mid R \models P \,\}.$$

This view makes clear that programs act by progressively narrowing the set of layouts. Adding a new constraint or directive (c) can only reduce the denotation (*refinement*), and if $P \subseteq Q$, then every realization that satisfies Q also satisfies P (*anti-monotonicity*).

$$[\![P \cup \{c\}]\!] \subseteq [\![P]\!], \qquad\qquad P \subseteq Q \;\Rightarrow\; [\![Q]\!] \subseteq [\![P]\!].$$

In this way, the spatial semantics of a Spytial program is precisely the set of realizations that remain admissible once all of its constraints and directives have been enforced. An unsatisfiable program is one whose denotation is empty:

$$[\![P]\!] = \emptyset \iff \forall R \in \mathsf{R}_{\text{wf}}, \neg(R \models P).$$

A formalization of these theorems is included in the ◇ supplement's Lean formalization.

5 Constraint (Dis)Satisfaction

Given the translations in table 1, checking whether a layout exists reduces to solving for box and group-boundary coordinates that satisfy the emitted inequalities.

When a program contains only orientation, align, and size constraints, all inequalities coexist in a single linear system. This is a standard linear feasibility problem over $\mathbb{Q}$, solvable in polynomial time by off-the-shelf linear solvers. Group and cyclic constraints add *disjunctions* to this system: each introduces a finite set of alternative linear subsystems representing distinct geometric cases. As a result, the global satisfaction problem for a Spytial program is: find a choice of branch for every grouping and cyclic constraint such that the corresponding union of inequalities is jointly feasible. In general, deciding the satisfiability of such disjunctive systems is NP-hard [4, 24]. In the Spytial context, however, the branching factor is small and structured: each grouping constraint contributes at most 4 alternatives (above, below, left, right) for each box excluded by a group, and each cyclic constraint contributes at most n rotations for a path of length n.

This means that Spytial is able to handle both conjunctions and disjunctions within a single engine. Conjunctive constraints are added directly to the Lume/Kiwi [3, 28] incremental linear solver, while disjunctive ones are expanded into candidate subsystems explored by bounded-depth backtracking. At each step the solver tentatively commits to one alternative, checks feasibility incrementally, and backtracks on failure. This yields a lightweight form of disjunctive linear programming implemented entirely atop an incremental linear solver.

5.1 Identifying Infeasible Subsystems

When a Spytial program is unsatisfiable, the key question is *why*. In linear programming, such explanations take the form of an *irreducible infeasible subsystem (IIS)*—the analogue of an *unsatisfiable core* in SAT/SMT solving. (For consistency with the solvers Spytial uses, we use the term

IIS in this paper.) An IIS is a minimal subset of constraints that cannot be satisfied together, even though every proper subset is feasible.

Because Spytial constraints introduce disjunctions, the solver explores a tree of linear systems. Each disjunct defines a branch, and the program as a whole is unsatisfiable only when *all* branches fail. Explaining such failures therefore has two parts: identifying a branch, and explaining inconsistency within that branch.

We illustrate this using the following small example. Suppose we have atoms $A = \{0, 1, 2\}$. Say we want to (a) lay them out left-to-right in ascending order; (b) align them horizontally; but also (c) group exactly $\{0, 2\}$. Because Spytial groups are rectangles (§4.2.3), the other constraints mean we cannot gerrymander a box that surrounds atom 0 and atom 2 while avoiding atom 1.

While it is easy to *discover* this unsatisfiability, presenting a diagnostic is much more tricky. An IIS can only be computed within a *single* linear system (i.e., once we have chosen a branch in the disjunction). The group constraint here introduces 4 such branches. Each branch defines its own feasibility problem, and only within a chosen branch does the concept of an IIS apply.

Selecting a representative branch. The solver must choose a branch of the disjunctive search tree that captures the conflict. It is unclear whether the most specific (deepest) or most general (shallowest) disjunction offers the most informative explanation. We have not been able to find work in disjunctive or linear constraint solving that has studied these human factors questions. Spytial therefore adopts a pragmatic strategy: it reports conflicts from the final failing branch in the solver's search, since that branch is both a concrete witness to infeasibility and already in hand when the solver terminates, requiring no additional exploration or bookkeeping. In this case, due to constraint structure, it will be the branch where atom 1 is to the *left* of the group boundary.

Isolating the conflicting constraints. Once a branch is fixed, the solver must identify the smallest subset of inequalities within that branch that together form the contradiction. Spytial extracts a quasi-IIS using iterative deletion: starting from the full conflicting set, it removes constraints while infeasibility persists. This deletion is aggressive for inequalities purely between boxes, but more conservative for those involving group bounding box constraints, using existing knowledge about group membership to avoid pruning essential constraints. The result is *subset-minimal*: no remaining constraint can be removed without restoring feasibility, though smaller infeasible subsets may still exist. In this case, the quasi-IIS is: 1 is to the left of the group boundary, 0 is to the left of 1, 1 is to the left of 2, 0 is horizontally aligned with 1, and 0 is horizontally aligned with 2.

Determinism. Given the same instance and specification, the solver follows a fixed iteration order over disjunctions and selects the same branch when relaxing constraints. Thus, the same invalid input produces the same quasi-IIS across runs. This provides predictability when debugging.

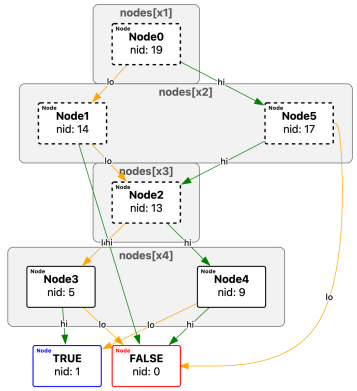
5.2 Reporting Unsatisfiable Diagrams

While identifying an IIS is crucial for diagnosing unsatisfiable Spytial programs, it is only half the battle. The other half is communicating this information effectively to users. Here, two strategies are common amongst diagramming systems:

Reject unsatisfiable layouts. Render diagrams only when the constraints are satisfiable and report a textual IIS otherwise. This could be shown either in terms of actual solver-level inequalities, or in terms of diagram elements as in Cope-and-Drag [32].

Produce a best-effort diagram. Penrose [51] produces a best-effort layout by *silently* weakening or dropping some constraints, not indicating which elements are in violation.

Both strategies are valuable. Users need to know *when* and *why* a conflict occurs, but a partial visualization is often more informative than just a textual report of conflicting constraints.



(a) Best-effort layout produced by weakening conflicting constraints; dotted boxes indicate IIS involvement.

Could not satisfy all constraints

Your data causes the following visualization constraints to conflict.

Source Constraints	Diagram Elements
`OrientationConstraint` with directions `[left]` and selector `{x,y:Node | x->y in lo and ...}`	`Node2 (nid: 13) left of Node1 (nid: 14)`
	`Node1 (nid: 14) left of Node0 (nid: 19)`
`OrientationConstraint` with directions `[right]` and selector `{x,y:Node | x->y in hi and ...}`	`Node5 (nid: 17) left of Node2 (nid: 13)`
	`Node0 (nid: 19) left of Node5 (nid: 17)`

(b) Textual report shown for conflicting constraints, both at the constraint and atom levels.

Fig. 6. Infeasibility report for the BDD diagram for formula $(\neg x_1 \wedge \neg x_2 \wedge ((x_3 \wedge \neg x_4) \vee (\neg x_3 \wedge x_4))) \vee (x_1 \wedge x_2 \wedge ((x_3 \wedge \neg x_4) \vee (\neg x_3 \wedge x_4)))$.

To illustrate this, consider our BDD example (§2). The astute reader will have noticed that our rules are subtly over-constrained. BDDs are directed acyclic graphs, not trees. Reduction merges isomorphic subgraphs and eliminates redundant nodes, so edges may have to skip levels in the variable ordering. As a result, it is possible to construct a BDD where it is impossible to maintain both strict layering by variable and uniform left/right orientation for low and high edges.

Figure 6 shows such a situation. Traditional solvers describe conflicts symbolically, but in a spatial layout they can be *situated*: each solver-level inequality corresponds to a geometric relation between boxes, each box to an atom, and each atom to a user-authored selector. Spytial therefore combines both approaches, reporting infeasibility at three connected levels:

Layout level: Produce a counterfactual diagram by relaxing all solver level inequalities in the IIS, and highlighting the boxes corresponding to variables involved (fig. 6a).

Diagram Element level: Express the conflict textually as concrete relations between atoms related to the variables involved in the IIS (fig. 6b, right).

Constraint level: Report the constraints behind the conflicting diagram-element relations (fig. 6b, left).

(Because it is built on the deterministic quasi-IIS, the same failure will always render the same diagram, which avoids confusing the viewer.) Together, these views connect solver-level infeasibility to both the spatial and semantic structure of the diagram. They let a diagrammer see not only *what* relationships conflict, but also *where* they occur and *why* they arise. Though the diagram is imperfect, it still conveys much of the structure of the BDD, allowing the diagrammer to better understand what went wrong. In §7.3, we evaluate whether this kind of situated reporting helps.

6 Integration with Languages

Spytial fundamentally needs to link its specification to the values created at run-time. How to do this depends very much on the details of individual language implementations. To explore this space, we have implemented Spytial for three very different kinds of languages. Critically, we added Spytial to each language without making *any* changes to the language implementation.

```
@orientation(
  selector='{x,y : RBNode | (y.key not in NoneType)
      and x.left = y}',
  directions=['left', 'below'])
@atomColor(
selector='{x: RBNode | @:(x.color) =
    red}', value='red')
class RBNode:
    ...
```

(a) Spytial in Python.

```
#[derive(Serialize,SpytialDecorators)]
#[orientation(
  selector="{x,y : RBNode | x->y in
      left}", directions=["left", "below"])]
#[atom_color(
  selector="{x : RBNode | @:(x.color) =
      Red}", value="red")]
struct RBNode {
    ...
}
```

(b) Spytial in Rust.

```
data RBNode:
  | Black(value, left, right)
  | Red(value, left, right)
  | Leaf
sharing:
method _output(self):
x = DR.genlayout(self,```
constraints:
  - orientation:
      selector: "{x, y : Red+Black | x->y in left}"
      directions:
        - left
        - below
directives:
  - atomColor:
      selector: Red
      value: red
  ```)
  VS.vs-constr-render("RBNode", [list:],
  { cli: lam(a) : a end, cpo: lam(a): x end })
  end,
end
```

(c) Spytial in Pyret.

Fig. 7. Spytial in three languages.

We chose *Python* because its highly dynamic nature and rich annotation framework makes it easy to attach information to data and propagate it to run-time (at a cost of both time and space!); in some ways, it reflects a best-case situation of what we can hope for in convenience when performance is not a concern. At nearly the other extreme we used *Rust*, which has a rich static type system, but tries to minimize run-time overhead. Finally, we chose *Pyret*, an educational programming language used to teach data structures at several institutions. Pyret is primarily a dynamic language, but of a parsimonious kind: it does not support arbitrary overloading, annotation, etc.

We provide brief code excerpts for red-black trees in fig. 7, showing a constraint and directive for each; the ◇ supplement contains full programs. The following subsections describe the mechanisms for Python, Rust, and Pyret in turn. Each one explains: how to obtain the data, how to recover structure, how to embed the Spytial specification, and how to invoke the diagram. We hope this variety of mechanisms offers concrete guidelines for Spytial integration into many more languages.

### 6.1 Python Integration

**Standard presentation of data.** Python presents values primarily through textual representations. Objects, lists, and dataclasses are shown as nested text, often via pretty printers in notebooks or libraries such as `pprint` and `dataclasses`. Tools like Pandas, Matplotlib, and Seaborn extend this to tabular or graphical plots, but these are designed for numerical or statistical datasets rather than structured program values.

**Recovering structure for visualization.** Python exposes object structure through reflection. Spytial uses this mechanism to walk dataclasses and containers (lists, dictionaries, etc.), mapping their fields and contents into explicit relational form. Each field in a dataclass yields a relation, and
```

each instance contributes atoms for its subvalues. This traversal happens entirely at runtime, using standard inspection mechanisms to walk the object graph and collect type information.

Describing spatiality. Spytial code is attached using decorators and runtime registration functions. Decorators add structured records to classes or instances describing how their values should appear in a diagram. During traversal, these records are merged along the method-resolution order so that class-level defaults, inherited specifications, and per-instance overrides compose cleanly. This keeps spatial specifications close to the data definitions they describe.

Presentation of diagrams. Spytial diagrams are generated through an explicit call in Python. After constructing a value, such as `RBNode(...)`, users must invoke `diagram(...)` to render the corresponding visualization. Evaluating a value in a REPL or printing it with `print(...)` produces only the standard textual form. This explicit trigger reflects Python's imperative style: rendering is a deliberate side effect in the current output context, such as a notebook cell or terminal.

Language-specific considerations. Spytial leverages `IPython`'s [29] rich display system to render diagrams inline alongside code and textual output. All examples in §7.1 are implemented in notebooks, demonstrating how Spytial integrates naturally into Python's interactive workflow.

One important consideration when working with Python is its heavy dependence on foreign-function libraries. Python often serves as a high-level gateway to systems implemented in other languages, ranging from numeric codes in Fortran to deep-learning frameworks in CUDA and C to solvers like Z3. The data from these systems can be opaque to Spytial, which does not (currently) have the ability to traverse the foreign stack and use the FFI's reflective features. Spytial instead accommodates such cases by allowing developers to register custom "relationalizers", which enable authors to take control over how values are decomposed into atoms and relations.

6.2 Rust Integration

Standard presentation of data. Rust presents values textually via `Debug`/`Display` and tool specific visualizers (e.g., IDE debuggers). These are linear or tree-shaped; relationships such as sharing are implicit.

Recovering structure for visualization. Rust's static type discipline precludes runtime reflection, but every value remains traversable through type-directed serialization. Spytial builds on this mechanism by interposing on the `serde::Serialize` trait [46], reusing its traversal to recover the structural relationships implicit in data definitions. Rather than emitting bytes, the serialization pass becomes a source of relational information from which Spytial constructs diagrams that mirror a value's relational structure.

Describing spatiality. Spytial operations are implemented as procedural macros that execute at compile time. By analyzing annotated type definitions, these macros collect and compose specifications across the type tree and embed them into the same type-directed serialization path. This treats spatial specification as a static, type-level elaboration rather than a runtime extension.

Presentation of diagrams. Spytial diagrams in Rust are generated through an explicit call. After constructing a value, users must invoke `diagram(...)` to render the corresponding visualization. Because Rust has no interactive REPL or native display surface, Spytial renders output externally, opening a browser window or writing an HTML file.

Language-specific considerations. Rust's compiled execution model provides portability and reproducibility: the same visualization can be generated from binaries, tests, or documentation builds. The cost of this design is a loss of co-location: diagrams appear outside the code context that produced them, unlike the inline renderings seen in Python notebooks or the Pyret REPL.

Table 2. Constraint/directive counts for CLRS-inspired structures. Entries beginning with + indicate additions relative to their base structure.

Structure	#Constr	#Dir	Structure	#Constr	#Dir
Max Heap	3	2	Memoization Matrix	5	1
Stacks	2	3	Huffman Codes (Tree)	5	2
Queues	2	5	B-Trees	5	3
Linked Lists	2	0	Fibonacci Heaps	4	5
Doubly Linked Lists	4	2	vEB Tree	2	3
Circular Linked Lists	2	2	Disjoint Sets (Forests)	6	2
Direct-Address Tables	3	4	**Unweighted Graph (Adj. Mat.)**	1	1
Chained Hash Tables	5	4	Topologically Sorted	+1	+0
Binary Search Trees	3	4	Highlighted SCCs	+1	+0
Red-Black Trees	3	5	**Weighted Graph (Adj. List)**	1	2
Order-Stat. Tree	+0	+1	Highlighted MST	+1	+2
Interval Tree	+0	+3			

6.3 Pyret Integration

Standard presentation of data. Pyret displays values through structured, interactive renderings in its browser-based IDE. By default, values appear in constructor notation that mirrors their algebraic datatype definitions. Many built-in types also offer alternate views (e.g., rationals can be viewed as decimals or fractions) by clicking on them.

Recovering structure for visualization. Rather than flattening the output into a single string, Pyret exposes values through an algebraic datatype known as the *value skeleton*. The value skeleton is a recursive datatype; this enables data at every level to be clickable to obtain other views. Spytial simply reuses this mechanism: constructors determine relations, and recursive descent enumerates atoms and their connections.

Describing spatiality. Pyret lacks language features such as annotations, macros, or runtime reflection. Instead, values can define their own rendering behavior by implementing an `_output` method (which is a generalization of Java's `toString`: the output can be any datatype that the IDE can render, which includes rich JavaScript output). Spytial builds on this mechanism: specifications, written as YAML strings, are associated with values through their `_output` definitions. Spytial interprets these specifications together with the corresponding Pyret values to produce diagrams.

Presentation of diagrams. Because Spytial integrates with the language's output mechanism, evaluating a value in the REPL—e.g., typing `Black(...)`—invokes its `_output` method, which renders the diagram inline automatically without needing further function calls.

Language-specific considerations. As noted, built-in renderers for compound data display sub-values using the same viewing mechanism. Thus each sub-value can be manipulated independently (e.g., clicked on to see other renderings). In an ideal world, this recursive process would continue through a Spytial diagram: each value rendered inside the diagram would be a Pyret DOM node with all of its views available. Unfortunately, this has been difficult to implement reliably inside SVGs. Thus, once a value is displayed as a diagram, its sub-values render in the default Pyret way.

7 Evaluation

Our evaluation examines three very distinct components: expressiveness (§7.1), running time (§7.2), and the value of counterfactual diagrams (§7.3). We also present preliminary use data (§7.4).

7.1 Evaluating Expressiveness

How do we evaluate the expressiveness of Spytial? Lacking a formal "benchmark", it is best if we could use an authoritative, externally-defined catalog. In §2 we used a diagram from Wikipedia, but Wikipedia can be uneven in its coverage and quality. Instead, we chose *Introduction to Algorithms* (CLRS) [11], an influential algorithms textbook known for its high coverage and careful, professionally-drawn diagrams, which reflect broadly accepted conventions for how data structures are presented and taught. Furthermore, a programmer familiar with these conventions might reasonably expect a similar structural presentation when visualizing their own runtime data structures.

We use these data structure diagrams as a test of expressiveness: whether Spytial can capture and apply, across arbitrary instances, the visual conventions used in CLRS to present these structures. In other words, this evaluation asks whether Spytial can reproduce key aspects of their structural organization across instances. (This does not, however, ensure that its generated diagrams match or exceed hand-designed textbook figures in overall quality or utility.)

Our selection spans the major parts of CLRS that present data structures through explicit diagrams: Chapter 6 (Heaps); Part III on Data Structures (Chapters 10–14); Part IV on Advanced Design and Analysis Techniques (Chapters 15–16); Part V on Advanced Data Structures (Chapters 18–21); and Part VI on Graph Algorithms (Chapters 22–23). Other chapters in the book are less data-structures-focused: analytic preliminaries (Chapters 1–5), sorting (Chapters 7–9), general algorithmic techniques (Chapters 16–17), algorithm-only graph material (Chapters 24–26), and "Selected Topics" (Part VII), which survey domain-specific material.

If we had programmed manually, we might have inadvertently injected Spytial-friendly biases. Therefore, we instead prompted two large-language models (GitHub Copilot using GPT-5 mini and Claude Sonnet 4.5) to generate Python implementations "in the style of CLRS", and manually reviewed the output for a sense of conformance. These implementations likely reflect patterns present in the code on which these models were trained, and thus provide a plausible approximation of existing implementation practice.[1]

We then annotated these LLM-generated programs with Spytial specifications sufficient to realize the corresponding CLRS visual language. Each specification uses only the built-in constraint and directive vocabulary (orientation, group, cyclic, attribute), without manual coordinate placement or solver extensions. Augmented structures (e.g., order-statistic trees extending red-black trees) reuse existing specifications by layering additional constraints or directives, demonstrating how new diagram specifications can compositionally build on existing ones.

Table 2 lists the number of constraints and directives used to diagram each data structure. Across all 22 examples, Spytial was able to realize the structure of the corresponding CLRS diagram using at most 11 total operations. Most required three to five constraints and two to four directives, while composite cases were simplified by layering new constraints atop existing ones. Unlike static figures in the source textbook, each specification yields an interactive diagram that generalizes to arbitrary data values. The full catalog of these, with examples, is given in the ◇ supplement.

7.2 Performance

We measured diagram generation times (combining constraint-solving and rendering: i.e., "user experience time") for each data structure in table 2. We generated random instances of sizes 5, 10, 25, and 50;[2] the limit of 50 is inspired by work such as Yoghourdjian et al. [52], which shows that

[1] Some implementations required a consistent adjustment. In the book, values like `None` or small integers can appear multiple times in a structure (e.g., several empty array slots). In Python, however, they are singletons, which prevents distinct visual instances. To reproduce the book, we wrapped these in lightweight objects so that each occurrence would be distinct.

[2] The memoization matrices need triangular numbers, so we used the closest corresponding sizes (6, 10, 28, 55).

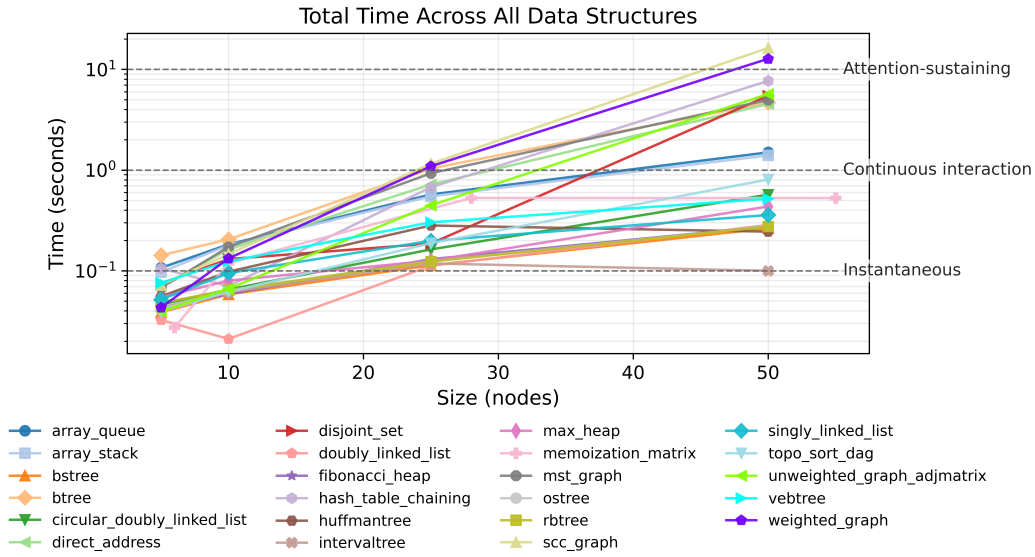


Fig. 8. Total time taken to render each CLRS data structure at various sizes, averaged over 20 runs on an Apple MacBook Pro (M1, 16 GB RAM). Nielsen response time thresholds are indicated by dashed lines.

human performance drops sharply around this size. Each configuration was executed twenty times on an Apple MacBook Pro (M1, 16 GB RAM). We report mean and standard deviation for all runs in the ◇ supplement. Figure 8 contextualizes this timing data using Jakob Nielsen's response-time labels for interactive systems [26].

(1) **Small diagrams ($N \leq 10$).** Most diagrams render in under 0.1 s, in the *instantaneous* range.
(2) **Medium diagrams ($N = 25$).** Most diagrams render in under 1 s, in the *continuous-interaction* range, where feedback remains perceptually smooth and uninterrupted.
(3) **Large diagrams ($N = 50$).** All but two diagrams complete within ten seconds, inside Nielsen's *attention-sustaining* range. It is worth noting that 50-element diagrams tend to be visually crowded and difficult to interpret except if looking for broad patterns.

The small dips are mainly due to measurement noise, and are accentuated by the log scale.

Naturally, the complexity theory limits (e.g., NP-hardness) of some of our constraint-solving tasks limit the usability of Spytial on very large data. However, at the sizes that humans are likely to find effective, implementation details are likely to matter more. Our implementation is still preliminary, and we did not optimize it further because we were obtaining sufficient performance on the above sizes. However, there is definitely room for improvement.

7.3 Unsatisfiable Constraints Study

§5.2 shows how Spytial reports an IIS through textual summaries and best-effort diagrams. To assess whether these representations help users interpret and debug unsatisfiable constraint sets, we conducted a between-subjects study comparing how users interpret unsatisfiable constraints under different feedback conditions. The ◇ supplement has the full instruments and details.

Participants ($N = 45$) were recruited from Prolific's [34] "computer programming skills" pool, compensated USD 5, and completed the task, on average, in 15 minutes. The study falls outside our IRB's area; we nevertheless took reasonable safeguards to protect the participants.

Table 3. Mean accuracy (%) and response time (s) by feedback condition ($N = 45$). Response time was analyzed with one-way ANOVA and, when significant, pairwise t-tests. Values in red are significantly slower.

	A: Text		B: Diagram		C: Both	
Question	Acc.	Time	Acc.	Time	Acc.	Time
Q1: Rule Identification	7% (1/14)	190s	6% (1/16)	193s	7% (1/15)	177s
Q2: Data comprehension	64% (9/14)	112s	94% (15/16)	35s	73% (11/15)	51s
Q3: Data change	57% (8/14)	185s	50% (8/16)	127s	40% (6/15)	168s

Participants were introduced to the constraint language through a familiar, visually structured domain: *family trees*. Each task began with a table specifying people, their ages, and parent-child relationships, followed by a short rule set describing how those relationships should be diagrammed.

Participants first completed a short training task involving a family structure that satisfied all layout rules, allowing them to learn how the rules map to spatial layouts. They were then randomly assigned to one of three feedback conditions. With respect to the elements of fig. 6, condition (A) saw only the Diagram Elements presentation of the IIS ($N = 14$), (B) only the best-effort diagram ($N = 16$), and (C) both ($N = 15$). All were asked three multiple-choice questions about a plausible family structure that *did not* satisfy layout rules:

(1) Identify which layout rules could not be satisfied for the given family.
(2) A question about the underlying data ("who is X's grandparent?").
(3) Describe how the data could be modified to satisfy all diagramming rules.

Table 3 summarizes accuracy and response times across feedback conditions. Accuracy did not differ significantly for any question (all $\chi^2 = 0.010, p = 0.995$).

For Q1 (rule identification), participants in all conditions struggled to identify which constraint rules caused unsatisfiability; only three successfully connected element-level feedback to higher-level rules. This highlights the need for the Source Constraints reporting in fig. 6.

For Q2 (data comprehension), a one-way ANOVA found a significant main effect of condition ($F(2, 42) = 6.99, p = 0.002$). Pairwise t-tests showed that participants that received textual feedback alone (A) were significantly slower than those shown a best-effort diagram (B, $p = 0.005$) and those shown both (C, $p = 0.032$). This supports the value of best-effort diagrams in helping users interpret data even when layout constraints are unsatisfiable.

Roughly half of participants answered Q3 (data modification) correctly, with no reliable differences across conditions ($\chi^2 = 0.864, p = .649$). It is possible that counterfactual reasoning (figuring out how to alter the data to satisfy all rules) depends more on domain knowledge than visualization, or at least lies outside the scope of our study constraints.

We note that these performances should be interpreted in the context of the user's goal. When a user is authoring the Spytial specification, tasks like Q1 are very important for fixing errors. But when a user is only consuming a specification, it may matter much less.

7.4 Student Use

In Fall 2025, we were able to deploy an early version of Spytial for Pyret in a first-year data structures course at a US university. Since this predated the full integration, students had to call a function to generate graphical output. This in turn enabled us to log how often students did so. Spytial specifications were written by an author of this paper; students were not expected to write them. (As fig. 7c shows, doing so would be ungainly, and likely error-prone for first-year students.)

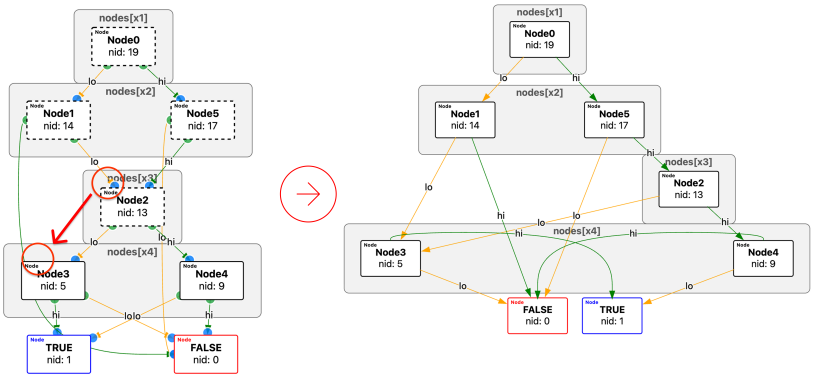


Fig. 9. Interactive editing of the BDD in fig. 6. Node1's lo edge is moved from Node2 to Node3, making the structure obey all constraints. The layout updates accordingly.

The class had 49 students. All logging was completely anonymized. Still, some students turned off logging, and many used browsers like Brave that block cookies, making it impossible for us to monitor their behavior. (Students were allowed to do both, to avoid interfering with their educational comfort.) University IRB policy allows anonymous reporting of student experience.

Across five assignments with Spytial support, a total of 26 distinct students (just over half the class) *observably* used it, making a total of 681 calls. (We actually logged 926 calls, but 245 were at such high frequency that we assumed they were automated uses and filtered them from our data.) The first two assignments with support had simple data; there were 77 calls from 13 users for the first, and only 12 from 5 for the second. The third, fourth, and fifth had richer tree- and graph-shaped data. For the third, we saw 91 calls from 10 users; for the fourth, 52 from 10; and for the last (graphs), 449 from 13. Most of these were repeat users, suggesting that when the data got interesting enough, they returned to use it.

Our weak monitoring makes it difficult to measure "popularity" (the above data are a lower bound: because Spytial runs entirely client-side, we have no other way to measure use). Much more useful is the performance data, which contextualize §7.2 with real human use. The mean number of atoms was 16.04 ($\sigma = 6.55$), with median 17. The mean "user experience" time was 1.3 s, but this was affected by a few large outliers: $\sigma = 2.7$ s. The median was only 130 ms, nearly in Nielsen's "instantaneous" range.

8 Interactive Value Construction

Sometimes it is useful to construct values graphically. This is especially handy when building or modifying non-trivial values for unit tests, for instance.

Because Spytial rules are declarative, the same rules that describe how a structure should *appear* can also dictate how it should *take shape* during construction. That is, every Spytial rule tells us about valid edits: what may be added, how it should appear, where it may go, and how existing parts should adjust.

We have implemented a prototype of this idea for Python. The underlying Spytial specification remains *unchanged*. User actions such as adding or removing atoms update the value under construction, and the layout solver incrementally re-satisfies all constraints. This produces immediate visual feedback: illegal placements are rejected, and valid ones snap into positions that preserve the declarative intent. Thus, values can be built under the same rules that make them readable. Space limits preclude a detailed explanation, but fig. 9 shows the tool in action.

9 Related Work

Spytial builds on long-standing observations from **cognitive science** that spatial organization supports reasoning. Work by Tversky [47, 48], Larkin and Simon [20], and the Gestalt tradition [19, 39, 49] shows that effective diagrams let users exploit spatial metaphors, see relational patterns as spatial structure, and keep together the information that must be used together [16, 23]. Spytial encodes these principles directly: its core constraints (e.g., alignment, grouping, and relative orientation) specify the spatial organization users rely on. Its directives (e.g., icon and atomcolor) provide controlled visual cues for conceptual categories.

The system's operations and the details of its layout engine also draw on a large body of work. Spytial users do not route edges. Instead, the layout engine applies principles—themselves also drawing on cognitive science and human factors—from **graph-drawing** [35, 43], especially those concerned with space efficiency, edge length, crossings, and bends. These practices reduce ambiguity and visual load, automatically yielding diagrams consistent with established best practices.

Spytial is also inspired by, and borrows from, several prior drawing systems:

Systems for Drawing. Frameworks such as D3.js [8] and TikZ [45] give authors fine-grained drawing control. They do so through imperative commands that place and style graphical primitives directly. This control, however, requires that diagrams be painstakingly constructed piece-by-piece through verbose, low-level operations [5, 25]. Diagramming also becomes an "all-or-nothing" proposition: users get no benefit until enough code has been written to produce a complete drawing. These systems also do not (and cannot) provide the kind of debugging Spytial provides. Authors must also code defensively against failure modes (e.g., malformed input data), but as Prasad et al. [32] argue, defensive design cannot protect against "unknown unknowns": failure modes that the author did not foresee. These silent failures can lead to misleading or incorrect diagrams.

Graph Description Languages. DOT [14], Mermaid [44], and PlantUML [37] are declarative languages that allow graphs and diagrams to be described in terms of nodes, edges, and attributes. They separate structure from layout: a specification encodes topology and style hints, while an external engine determines geometry. They enable convenient interchange and automatic rendering but offer limited spatial control: authors can suggest but not enforce spatial relationships. Such languages describe *what* to draw and provide only coarse influence over *how* it is arranged. The geometry itself is typically determined by a layout engine.

Layout Generation Systems. A large body of work studies *layout generation*: computing the geometry of a diagram once its graph structure is fixed. This includes constraint-based layout systems (e.g., WebCola [12, 13], Auto Layout [2, 3, 17]), fixed algorithmic layout engines (e.g., Graphviz [14], DAGRE [30]), and layout-algorithm synthesis systems (e.g., MEDEA [21]).

Constraint-based layout systems such as Auto Layout and WebCola support declarative geometric relationships between elements, including alignment, spacing, and other relative placement constraints. Their role, however, is to produce a layout rather than to treat these relationships as semantic requirements: when not all requested relationships can be satisfied simultaneously, the solver still returns a best-effort result. Consequently, such systems generally do not identify conflicting constraint sets or explain unsatisfiable specifications.

Graphviz and DAGRE instead implement fixed algorithmic pipelines for graph layout. Graphviz provides several layout engines, while DAGRE focuses on directed graphs using hierarchical layering, crossing minimization, and edge routing. In both cases, users may tune global parameters, but have limited ability to express instance-level spatial relationships between elements.

MEDEA addresses a different problem again: it synthesizes layout algorithms themselves from declarative specifications, with an emphasis on efficiency and incremental computation.

Spytial operates at a different layer of the diagramming stack. Rather than designing or synthesizing layout algorithms, it focuses on expressing *instance-level spatial constraints* over program data and diagnosing when those constraints cannot be satisfied. In our implementation, WebCola serves only as a backend for geometric realization once constraints have been validated.

Diagramming by Constraint. We finally compare to systems that also treat diagramming as a constraint specification problem. Spytial draws particular inspiration from three constraint-based diagramming systems: Penrose [51], Bluefish [31], and Cope-and-Drag [32]. Each treats layout declaratively, using constraints or relations rather than a sequence of drawing commands:

Penrose programs are divided into three parts: a *domain* defining the rules of a system, a *substance* describing an instance of those rules, and a *style* specifying how to render it. It generates diagrams by numerical optimization over soft constraints.

Bluefish takes a compositional view of diagramming: diagrams are constructed by composing local relations in the host language, rather than by solving a single global constraint problem. Abstraction over diagram structure is therefore delegated to the host language (e.g., JavaScript), and relational composition becomes a first-class part of diagram construction. This supports reusable components and locally composed layout relations in ordinary code.

Cope-and-Drag provides a small vocabulary of spatial constraints and directives for diagramming Alloy instances. Its design emphasizes a useful separation between *constraints*, which specify spatial relationships, and *directives*, which control presentation. This makes it a compact and declarative system for constraint-oriented diagramming in the Alloy setting, and one whose basic vocabulary Spytial extends. Its specifications are applied at a coarse granularity, however, over all elements of a particular type or relation.

Spytial's novelty lies in combining the following three properties for runtime program values:

(1) Specifications apply directly to program values and generalize across instances. Cope-and-Drag is tied to the Alloy setting, Penrose requires an explicit substance to be coupled with a style, and Bluefish is most naturally used to build individual diagrams in the host language rather than reusable families of layouts (e.g., a specific BDD rather than any BDD).
(2) Layouts have a well-defined, order-independent semantics. In Bluefish, the effect of a relation depends on local ownership and composition order, while in Penrose, authors often have to attach operations to particular optimization phases to obtain the desired behavior.
(3) Unsatisfiability is expressed both diagrammatically and in terms of the source language. As Prasad et al. [32, Fig. 16] show, Penrose's treatment of constraints as soft can yield misleading "best-effort" diagrams when constraints are unsatisfiable: violated relationships are absorbed as visual imperfections rather than surfaced as explicit inconsistency. When layouts are unsatisfiable, Cope-and-Drag instead produces no diagram, reporting only an unsatisfiable core (§7.3 discusses the limitations of this text-only view). Spytial combines the strengths of both approaches by preserving the contextual value of a diagram while also reporting, at the element and source-constraint levels, why the specification is inconsistent.

In addition, unlike the others, Spytial is formalized with proven metatheorems about its semantics. Together, these properties make it a reusable and debuggable diagramming system for program data rather than a standalone diagram generation tool.

10 Discussion

Generative AI (GenAI) Do we need Spytial? Why not just use GenAI instead? Setting aside the dismal future this would suggest for programming languages research, we believe there are several good reasons for Spytial even in this era.

It is not always straightforward to get GenAI to generate an accurate visualization. In our experience, it sometimes works out of the box, sometimes needs coaxing, sometimes leads to run-time errors, and sometimes (most frustratingly) only *partially* works and can't easily be fixed. Now the developer is stuck with our motivating problem—having to learn an unrelated skill—with both a codebase *and* potentially a framework they don't already know.

Even when everything works, it would take far more prompting to follow good visualization rules; obtain good debugging; support input; and make the diagrams interactive. Even if all this were done automatically, we would still need to tell the GenAI what spatial relationships to preserve during interaction. This last step is tantamount to…writing a Spytial program, which then does all the other steps automatically and predictably. Instead, we think Spytial might be a good *target* for GenAI. In addition, Spytial focuses on functionality, not prettiness. We can create prettier or richer output by enriching Spytial with hooks where GenAI (or a human) can insert custom visualization and interaction code.

Accessibility Fortunately, Spytial does not affect the baseline accessibility provided by the language (which is usually minimal, with a few exceptions [38, 40, 41]). Intriguingly, studies show that even congenitally blind people develop robust spatial awareness [22]. Since Spytial's constraints form a declarative specification of spatial structure, they could potentially be repurposed to support non-visual output—like sonic or haptic renderings—that preserve structural cues.

3D It should be possible to extend Spytial to 3D, which could aid data structures that render poorly in a plane. This would require enriching the language, adding constraints, and using 3D-renderers. Given the declarative nature of the language, the burden on users to employ this should be much smaller than in most or all of the alternatives discussed in §9.

Ghost Variables When proving specifications, it is sometimes necessary to add ghost variables or fields [27] for the purpose of completing the proof. It is natural to wonder if a similar phenomenon occurs to support diagramming. In this case the additions would not be "ghost" because they would need to be present in the run-time values. Although we did not need such additions for CLRS (§7.1), we did need them for an AVL-tree [1] visualization used in a class demo (§7.4), because the selector language—being first-order logic—is a poor match for counting, which is needed to compute node height. We therefore expect more cases like this, and programmers may need conditional-compilation-style features to avoid the space and time costs in production.

Algorithm Visualization It is natural to wonder whether Spytial can do at least rudimentary algorithm visualization [9], assuming it can be notified about relevant state changes. If only directives change (e.g., coloring nodes while traversing a fixed graph), it should work well. All other changes can potentially produce very different layouts, making it difficult to identify what really changed, in violation of long-established perceptual psychology principles [15]. Instead, we should make each output *as similar as possible* to the previous one. A promising approach is to treat each object's location as a *soft constraint* when laying out the next state, and trying to minimize its movement. We have, however, not yet experimented with this idea to find its weaknesses.

Acknowledgments

This work is partially funded by the US NSF grants 2227863 and 2208731 and Brown's UTRA. We thank Elijah Rivera, Gavin Gray, Ji Won Chung, Skyler Austen, Joe Gibbs Politz, Amal Ahmed, Hossein Haeri, James McKinna, Rob Goldstone, Wolfgang Gatterbauer, and the reviewers.

Data Availability Statement

All materials needed to reproduce the results in this paper are included in the accompanying artifact [33] (https://zenodo.org/records/19401958) and supplement (https://zenodo.org/records/19520876). The language embeddings described in sections 6.1 to 6.3 are available at https://github.com/sidprasad/spytial (Python), https://github.com/sidprasad/caraspace (Rust), and https://github.com/sidprasad/spyret-ide (Pyret). These materials also include the Lean mechanization of the spatial semantics described in §4, the Jupyter notebooks and benchmark data used in the evaluations in §s 7.1 and 7.2, and the survey instrument, the survey export analyzed in the paper, and the analysis script used to reproduce the reported aggregate statistics for the study in §7.3.

References

[1] G. M. Adel'son-Vel'skii and E. M. Landis. 1962. An Algorithm for the Organization of Information. *Soviet Mathematics Doklady* 3 (1962), 1259–1263.

[2] Apple Inc. 2016. Auto Layout Guide. https://developer.apple.com/library/archive/documentation/UserExperience/Conceptual/AutolayoutPG/index.html.

[3] Greg J Badros, Alan Borning, and Peter J Stuckey. 2001. The Cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)* 8, 4 (2001), 267–306.

[4] Egon Balas. 1979. Disjunctive programming. *Annals of discrete mathematics* 5 (1979), 3–51.

[5] Leilani Battle, Danni Feng, and Kelli Webber. 2022. Exploring D3 implementation challenges on Stack Overflow. In *2022 IEEE Visualization and Visual Analytics (VIS)*. IEEE, Oklahoma City, OK, USA, 1–5. doi:10.1109/VIS54862.2022.00009

[6] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. 2018. On the dichotomy of debugging behavior among programmers. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 572–583. doi:10.1145/3180155.3180175

[7] Dirk Beyer. 2005. Binary decision diagram (simple example). Wikimedia Commons, licensed under GFDL and CC BY-SA 2.5. https://commons.wikimedia.org/wiki/File:BDD_simple.svg

[8] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D^3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (2011), 2301–2309.

[9] Marc H. Brown. 1987. *Algorithm animation.* Ph. D. Dissertation. Department of Computer Science, Brown University, Providence, RI, USA. UMI Order No. GAX87-15461.

[10] Cburnett. 2007. Circular Buffer. Wikimedia Commons. CC BY-SA 3.0. https://commons.wikimedia.org/w/index.php?curid=2302964 By the original author. License: CC BY-SA 3.0.

[11] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3 ed.). The MIT Press, Cambridge, MA, USA.

[12] Tim Dwyer. 2017. cola.js: Constraint-Based Layout in the Browser. https://ialab.it.monash.edu/webcola/ Accessed: 2024-12-02.

[13] Tim Dwyer, Kim Marriott, and Michael Wybrow. 2009. Topology preserving constrained graph layout. In *Graph Drawing: 16th International Symposium, GD 2008, Heraklion, Crete, Greece, September 21-24, 2008. Revised Papers 16.* Springer, 230–241.

[14] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. 2004. Graphviz and Dynagraph — Static and Dynamic Graph Drawing Tools. In *Graph Drawing Software*, Michael Jünger and Petra Mutzel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 127–148. doi:10.1007/978-3-642-18638-7_6

[15] James J Gibson and Eleanor J Gibson. 1955. Perceptual learning: Differentiation or enrichment? *Psychological review* 62, 1 (1955), 32–41. doi:10.1037/h0048826 Publisher: American Psychological Association.

[16] Robert Goldstone. 1994. An efficient method for obtaining similarity data. *Behavior Research Methods, Instruments, & Computers* 26 (1994), 381–386.

[17] Ijzeren Hein. 2021. AutoLayout.js: Apple's Auto Layout and Visual Format Language for JavaScript. https://github.com/IjzerenHein/autolayout.js.

[18] Daniel Jackson. 2006. *Software abstractions: logic, language, and analysis.* MIT Press, Cambridge, Mass.

[19] Kurt Koffka. 1922. Perception: An Introduction to the Gestalt Theory. *Psychological Bulletin* 19 (1922), 531–585.

[20] Jill H Larkin and Herbert A Simon. 1987. Why a Diagram is (Sometimes) Worth Ten Thousand Words. *Cognitive science* 11, 1 (1987), 65–100. doi:10.1111/j.1551-6708.1987.tb00863.x Publisher: Elsevier.

[21] Junrui Liu, Yanju Chen, Eric Atkinson, Yu Feng, and Rastislav Bodik. 2023. Conflict-Driven Synthesis for Layout Engines. *Proc. ACM Program. Lang.* 7, PLDI, Article 132 (June 2023), 22 pages. doi:10.1145/3591246

[22] Jack M Loomis, Roberta L Klatzky, Reginald G Golledge, Joseph G Cicinelli, James W Pellegrino, and Phyllis A Fry. 1993. Nonvisual navigation by blind and sighted: assessment of path integration ability. *Journal of Experimental Psychology:*

General 122, 1 (1993), 73–91. doi:10.1037/0096-3445.122.1.73 Publisher: American Psychological Association.

[23] Dor Ma'ayan, Wode Ni, Katherine Ye, Chinmay Kulkarni, and Joshua Sunshine. 2020. How Domain Experts Create Conceptual Diagrams and Implications for Tool Design. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/3313831.3376253

[24] Ashutosh Mahajan and Ted Ralphs. 2010. On the complexity of selecting disjunctions in integer programming. *SIAM Journal on Optimization* 20, 5 (2010), 2181–2198.

[25] Lekha Nair, Sujala Shetty, and Siddhanth Shetty. 2016. Interactive visual analytics on Big Data: Tableau vs D3.js. *Journal of e-Learning and Knowledge Society* 12, 4 (2016), 139–150. doi:10.20368/1971-8829/1128 Publisher: Italian e-Learning Association.

[26] Jakob Nielsen. 1994. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[27] Susan Owicki and David Gries. 1976. An axiomatic proof technique for parallel programs I. *Acta Informatica* 6, 4 (Dec. 1976), 319–340. doi:10.1007/BF00268134

[28] Joseph Orbegoso Pea and contributors. 2024. Kiwi: Fast TypeScript implementation of the Cassowary constraint solving algorithm. https://github.com/lume/kiwi. Version 0.4.2, accessed 2025-11-12.

[29] Fernando Perez and Brian E. Granger. 2007. IPython: A System for Interactive Scientific Computing. *Computing in Science & Engineering* 9, 3 (2007), 21–29. doi:10.1109/MCSE.2007.53

[30] Chris Pettitt and contributors. 2014. Dagre: A JavaScript library for directed graph layouts. https://github.com/dagrejs/dagre. Accessed: 2024-11-21.

[31] Josh Pollock, Catherine Mei, Grace Huang, Elliot Evans, Daniel Jackson, and Arvind Satyanarayan. 2024. Bluefish: Composing Diagrams with Declarative Relations. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology* (Pittsburgh, PA, USA) *(UIST '24)*. Association for Computing Machinery, New York, NY, USA, Article 23, 21 pages. doi:10.1145/3654777.3676465

[32] Siddhartha Prasad, Ben Greenman, Tim Nelson, and Shriram Krishnamurthi. 2025. Lightweight Diagramming for Lightweight Formal Methods: A Grounded Language Design. In *39th European Conference on Object-Oriented Programming (ECOOP 2025) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 333)*, Jonathan Aldrich and Alexandra Silva (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 26:1–26:29. doi:10.4230/LIPIcs.ECOOP.2025.26

[33] Siddhartha Prasad, Michael Tu, Karan Kashyap, Tim Nelson, and Shriram Krishnamurthi. 2026. Artifact For: Diagramming Program Values by Spatial Refinement. doi:10.5281/zenodo.19401958

[34] Prolific. 2025. Prolific. https://www.prolific.com. London, UK. Accessed April 2025.

[35] Helen Purchase. 1997. Which aesthetic has the greatest effect on human understanding?. In *International Symposium on Graph Drawing*. Springer, Berlin, Heidelberg, 248–261. doi:10.1007/3-540-63938-1_67

[36] Radke7CB. 2022. Binary tree v2.svg. Wikimedia Commons, CC BY-SA 4.0. https://commons.wikimedia.org/wiki/File:Binary_tree_v2.svg

[37] Arnaud Roques. 2025. *PlantUML*. PlantUML Project. https://plantuml.com/ Diagramming software.

[38] Emmanuel Schanzer, Sina Bahram, and Shriram Krishnamurthi. 2020. Adapting Student IDEs for Blind Programmers. In *Proceedings of the 20th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '20)*. Association for Computing Machinery, New York, NY, USA, Article 23, 5 pages. doi:10.1145/3428029.3428051

[39] Lothar Spillmann and Max Wertheimer. 2012. *On Perceived Motion and Figural Organization*. The MIT Press. doi:10.7551/mitpress/9222.001.0001 Includes English translations of Max Wertheimer's 1912 and 1923 papers on Gestalt psychology.

[40] Andreas Stefik, Andrew Haywood, Shahzada Mansoor, Brock Dunda, and Daniel Garcia. 2009. SODBeans. In *IEEE 17th International Conference on Program Comprehension*. IEEE, 293–294. doi:10.1109/ICPC.2009.5090064

[41] Andreas Stefik and Richard Ladner. 2017. The Quorum Programming Language. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 641. doi:10.1145/3017680.3022377

[42] J Ridley Stroop. 1935. Studies of interference in serial verbal reactions. *Journal of experimental psychology* 18, 6 (1935), 643.

[43] Kozo Sugiyama. 2002. *Graph Drawing and Applications for Software and Knowledge Engineers*. Vol. 11. World Scientific, Singapore.

[44] Knut Sveidqvist and Ashish Jain. 2021. *The Official Guide to Mermaid.js*. Packt Publishing Ltd, Birmingham, England.

[45] Till Tantau. 2013. Graph Drawing in TikZ. In *Graph Drawing*, Walter Didimo and Maurizio Patrignani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 517–528.

[46] David Tolnay. 2025. *Serde*. https://serde.rs/ Serialization framework for Rust.

[47] Barbara Tversky. 2001. Spatial Schemas in Depictions. In *Spatial Schemas and Abstract Thought*. The MIT Press. doi:10.7551/mitpress/6392.003.0006

[48] Barbara Tversky and Masaki Suwa. 2009. *Thinking with Sketches.* Oxford University Press, Chapter 4, 75–84. doi:10.1093/acprof:oso/9780195381634.003.0004 _eprint: https://academic.oup.com/book/0/chapter/152650177/chapter-ag-pdf/44969773/book_7628_section_152650177.ag.pdf.

[49] Max Wertheimer. 1912. Experimentelle Studien über das Sehen von Bewegung. *Zeitschrift für Psychologie* 61 (1912), 161–265.

[50] Benjamin W White. 1969. Interference in identifying attributes and attribute names. *Perception & Psychophysics* 6 (1969), 166–168.

[51] Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. 2020. Penrose: from mathematical notation to beautiful diagrams. *ACM Trans. Graph.* 39, 4, Article 144 (Aug. 2020), 16 pages. doi:10.1145/3386569.3392375

[52] Vahan Yoghourdjian, Yalong Yang, Tim Dwyer, Lee Lawrence, Michael Wybrow, and Kim Marriott. 2021. Scalability of Network Visualisation from a Cognitive Load Perspective. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (2021), 1677–1687. doi:10.1109/TVCG.2020.3030459

Received 2025-11-10; accepted 2026-04-03